NodeJS

Revin

目　录

- gulp-minify-css 压缩css插件
- gulp-sass 将sass预处理为css
- gulp-less 将less预处理为css
- gulp-sourcemaps 插件
- gulp-concat 合并插件
- gulp-uglify 压缩JS插件
- gulp-util gulp常用工具库插件
- yargs插件
- gulp-nodemon 自动启动/重启插件
- coffee-script 插件
- gulp-coffee插件
- gulp-livereload 网页自动刷新

Moment.js-处理时间插件
express 前端框架
Async-异步流程控制插件
node-progress进度条插件
JSHint-代码规范检查工具
lodash -JavaScript 工具库
资料
框架所用包
bodyParser

简介

> 使用 Node.js 的优势和劣势都有哪些？

> 深入浅出Node.js（一）：什么是Node.js

> 多线程有什么用？

> Github 编程语言分布统计：JavaScript 称霸

Node简介

客户端的JavaScript是怎样的

- 什么是 JavaScript？
  - 脚本语言
  - 运行在浏览器中
  - 一般用来做客户端页面的交互（Interactive）
- JavaScript 的运行环境？
  - 运行在浏览器内核中的 JS 引擎（engine）
- 浏览器中的 JavaScript 可以做什么？
  - 操作DOM（对DOM的增删改、注册事件）
  - AJAX/跨域
  - BOM（页面跳转、历史记录、console.log()、alert()）
  - ECMAScript
- 浏览器中的 JavaScript 不可以做什么？
  - 文件操作（文件和文件夹的CRUD）
  - 没有办法操作系统信息
  - 由于运行环境特殊
- JavaScript 只可以运行在浏览器中吗？
  - 不是
  - 能运行在哪取决于，这个环境有没有特定平台

什么是Node

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。Node.js 使用了一个事件驱动、非阻塞式 I/O 的模型，使其轻量又高效。Node.js 的包管理器 npm，是全球最大的开源库生态系统。

通俗解释；

- Node 就是 JavaScript 语言在服务器端的运行环境
- 所谓“运行环境（平台）”有两层意思：
  - 首先，JavaScript 语言通过 Node 在服务器运行，在这个意义上，Node 有点像 JavaScript 虚拟机；
  - 其次，Node 提供大量工具库，使得 JavaScript 语言与操作系统互动（比如读写文件、新建子进程），在这个意义上， Node 又是 JavaScript 的工具库。

###注意：

- 是Node选择了JavaScript，不是JavaScript发展出来了Node。
- Node是一个JavaScript的运行环境（平台），不是一门语言，也不是JavaScript的框架

配置安装

window

window

window环境配置

安装包的方式安装

- 安装包下载链接：
  - Mac OSX： darwin
  - Windows：
    - x64
    - x86
- 安装操作：
  - 一路Next

更新版本

- 操作方式：
  - 重新下载最新的安装包；
  - 覆盖安装即可；
- 问题：
  - 以前版本安装的很多全局的工具包需要重新安装
  - 无法回滚到之前的版本
  - 无法在多个版本之间切换（很多时候我们要使用特定版本）

NVM工具的使用

> Node Version Manager（Node版本管理工具）

由于以后的开发工作可能会在多个Node版本中测试，而且Node的版本也比较多，所以需要这么款工具来管理

window安装操作步骤

1. 下载：nvm-windows
2. 解压到一个全英文路径
3. 编辑解压目录下的 `settings.txt` 文件（不存在则新建）

- `root 配置为当前 nvm.exe 所在目录`
- `path 配置为 node 快捷方式所在的目录`
- `arch 配置为当前操作系统的位数（32/64）`
- `proxy 不用配置`

```
root: C:\xxxxx\nvm
path: C:\xxxxxx\nodejs
arch: 64
proxy:
```

4. 配置环境变量 可以通过 window+r : sysdm.cpl

- `NVM_HOME = 当前 nvm.exe 所在目录`
- `NVM_SYMLINK = node 快捷方式所在的目录`
- `PATH += %NVM_HOME%;%NVM_SYMLINK%;`
- 打开CMD通过 `set [name]` 命令查看环境变量是否配置成功
- PowerShell中是通过 `dir env:[name]` 命令

5. NVM使用说明：

- https://github.com/coreybutler/nvm-windows/

6. NPM的目录之后使用再配置

配置Python环境

> Node中有些第三方的包是以C/C++源码的方式发布的，需要安装后编译
>
> 确保全局环境中可以使用python命令

环境变量的概念

> 环境变量就是操作系统提供的系统级别用于存储变量的地方

- Windows中环境变量分为系统变量和用户变量
- 环境变量的变量名是不区分大小写的
- 特殊值：
  - PATH 变量：只要添加到 PATH 变量中的路径，都可以在任何目录下搜索

补充：Windows下常用的命令行操作

- 切换当前目录（change directory）：cd
- 创建目录（make directory）：mkdir
- 查看当前目录列表（directory）：dir
  - 别名：ls（list）
- 清空当前控制台：cls
  - 别名：clear

- 删除文件：del
  - 别名：rm

> 注意：所有别名必须在新版本的 PowerShell 中使用

###资料

管理 node 版本，选择 nvm 还是 n？

基础知识

Node 命令基本用法

##Node 命令基本用法

进入 REPL 环境：

```
$ node
```

执行脚本字符串：

```
$ node -e 'console.log("Hello World")'
```

运行脚本文件：

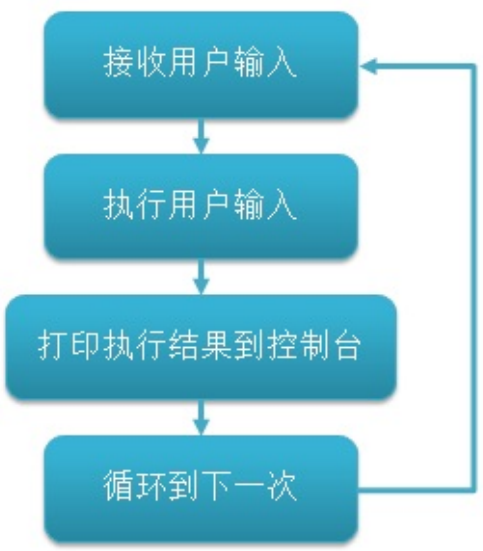
```
$ node index.js
$ node path/index.js
$ node path/index
```

查看帮助：

```
$ node --help
```

##REPL 环境

- REPL全称（Read，Eval，Print，Loop）



##REPL 环境操作

- 进入REPC:
  - node
  - node --use_strict (严格模式)
- 环境中
  - 可使用console命令
  - 特殊变量下划线 (_) 表示上一个命令的返回结果
  - .exit 退出

进程和线程

##进程

- 每一个 正在运行 的应用程序都称之为进程。
- 每一个应用程序运行都至少有一个进程
- 进程是用来给应用程序提供一个运行的环境
- 进程是操作系统为应用程序分配资源的一个单位

##线程

- 用来执行应用程序中的代码
- 在一个进程内部，可以有很多的线程
- 在一个线程内部，同时只可以干一件事
- 而且传统的开发方式大部分都是 I/O 阻塞的
- 所以需要多线程来更好的利用硬件资源
- 给人带来一种错觉：线程越多越好

##什么原因让多线程没落

- 多线程都是假的，因为只一个 CPU（单核）
- 线程之间共享某些数据，同步某个状态都很麻烦
- 更致命的是：
    - 创建线程耗费
    - 线程数量有限
    - CPU 在不同线程之间转换，有个上下文转换，这个转换非常耗时

调试

##Node 调试

- 最方便也是最简单的：console.log()
- Node 原生的调试

  https://nodejs.org/api/debugger.html
- 第三方模块提供的调试工具

```
$ npm install node-inspector –g
$ npm install devtool -g
```

- 开发工具的调试

  推荐 Visual Studio Code(1:设置断点 2:F5 3：选择Nodejs 4:配置调试文件[program])

  WebStorm

异步、回调

##异步操作

- Node 采用 Chrome V8 引擎处理 JavaScript 脚本，V8 最大特点就是单线程运行，一次只能运行一个任务。
- Node 大量采用异步操作（asynchronous operation），即任务不是马上执行，而是插在任务队列的尾部，等到前面的任务运行完后再执行。
- 提高代码的响应能力。
-

##异步操作回调

- 由于系统永远不知道用户什么时候会输入内容，所以代码不能永远停在一个地方；
- Node 中的操作方式就是以异步回调的方式解决无状态的问题；

全局对象

##全局对象

> https://nodejs.org/dist/latest-v4.x/docs/api/globals.html

（只有global和process和console是全局成员）

- global：

  类似于客户端 JavaScript 运行环境中的 window

- process：

  用于获取当前的 Node 进程信息，一般用于获取环境变量之类的信息

例子：

```
var argvs = process.argv.slice(2);

switch (argvs[0]) {
  case 'init':
    console.log('你需要INIT');
    break;
  case 'install':
    var installPackageName = argvs[1];
    console.log('你在安装' + installPackageName);
    break;
  case 'uninstall':
    console.log('uninstall');
    break;
}
```

- console：

  Node 中内置的 console 模块，提供操作控制台的输入输出功能，常见使用方式与客户端类似

例子:

```
var a = 1;

// 此处的Console是NODE平台提供的
console.log(a);

console.error(new Error('error'));
```

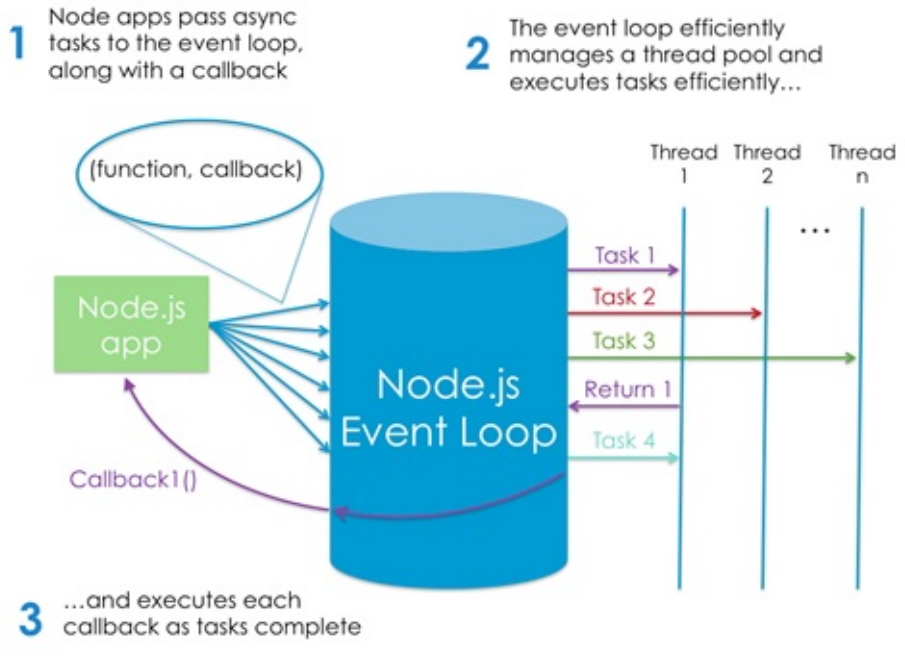
非阻塞 I/O和事件驱动和非阻塞机制

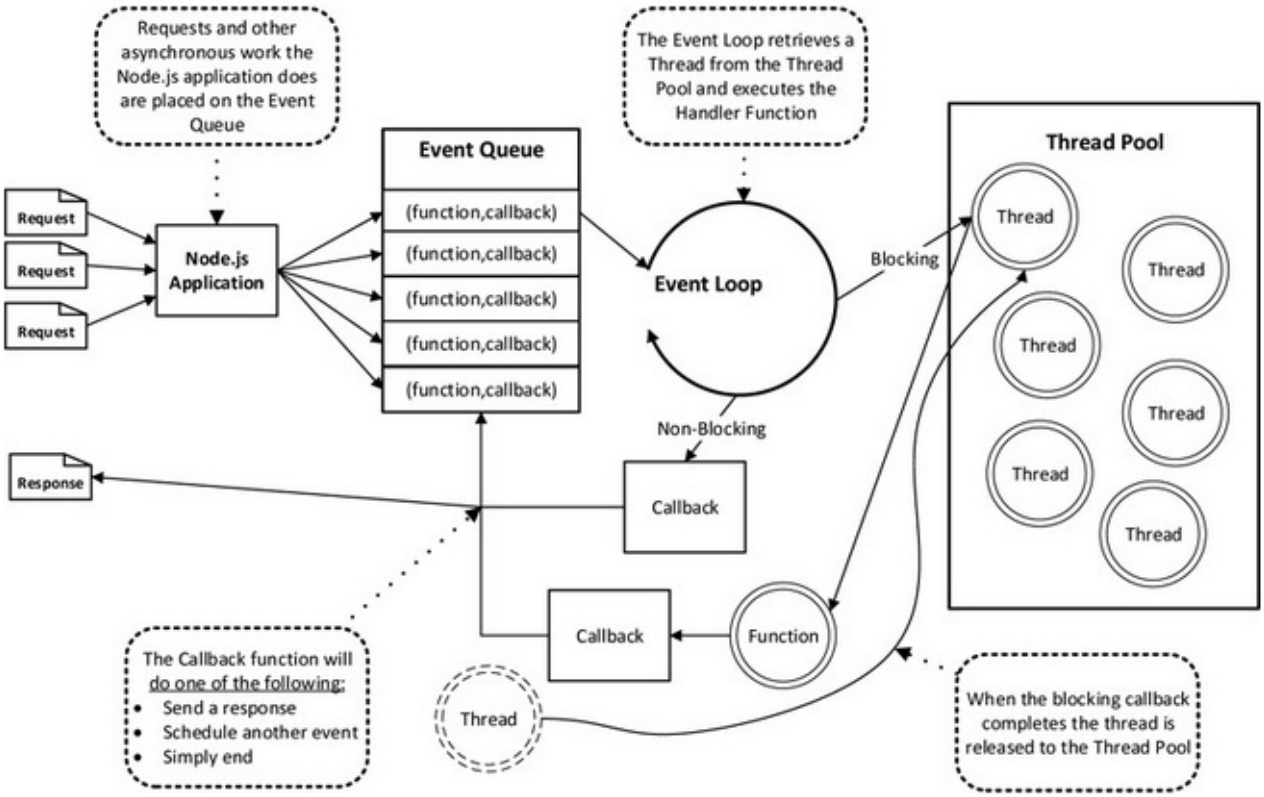
##什么是 I/O

I/O：【input/output】

可以理解为从输入到输出之间的转化过程

##事件驱动和非阻塞机制





- Node 中将所有的阻塞操作交给了内部实现的线程池
- Node 本身主线程主要就是不断的往返调度

##资料

Node.js 事件循环

Nodejs的事件驱动模型

Node.js的事件驱动模型

##非阻塞的优势

- 提高代码的响应效率
- 充分利用单核 CPU 的优势
- 改善 I/O 的不可预测带来的问题

npm包概念

##什么是包

- 由于 Node 是一套轻内核的平台，虽然提供了一系列的内置模块，但是不足以满足开发者的需求，于是乎出现了包（package）的概念：
- 与核心模块类似，就是将一些预先设计好的功能或者说 API 封装到一个文件夹，提供给开发者使用；

##包的加载机制

- id: 包名的情况：require('http')
  - 先在系统核心（优先级最高）的模块中找；
  - 以后不要创建一些和现有的包重名的包；
  - 然后再到当前项目中 node_modules 目录中找；

##NPM 概述

- 随着时间的发展，NPM 出现了两层概念：
  - 一层含义是 Node 的开放式模块登记和管理系统，亦可以说是一个生态圈，一个社区
  - 另一层含义是 Node 默认的模块管理器，是一个命令行下的软件，用来安装和管理 Node 模块。
- 官方链接： https://www.npmjs.com/
- 国内加速镜像： https://npm.taobao.org/
- 可以通过 NRM： Node Registry Manager

##安装 NPM

- NPM 不需要单独安装。默认在安装 Node 的时候，会连带一起安装 NPM。
- 但是，Node 附带的 NPM 可能不是最新版本，最好用下面的命令，更新到最新版本。
  `$ npm install npm -g`
- 默认安装到当前系统 Node 所在目录下。
- 由于之前使用 NVM 的方式安装的 Node 所以需要重新配置 NPM 的全局目录

##配置 NPM 的全局目录

- `$ npm config set prefix [pathtonpm]`
- 将 NPM 目录配置到其他目录时，必须将该目录放到环境变量中，否则无法再全局使用

##常用 NPM 命令

- https://docs.npmjs.com/

```
npm config [ls|list|set|get] [name] [value]
npm init [--yes|-y]
npm search [name]
npm info [name]
npm install [--global|-g] [name]
npm uninstall [--global|-g] [name]
npm list [--global|-g]
npm outdated [--global|-g]
npm update [--global|-g] [name]
npm run [task]
npm cache [clean]
```

模块化

##模块化代码结构

- Node 采用的模块化结构是按照 CommonJS 规范
- 模块与文件是一一对应关系，即加载一个模块，实际上就是加载对应的一个模块文件。

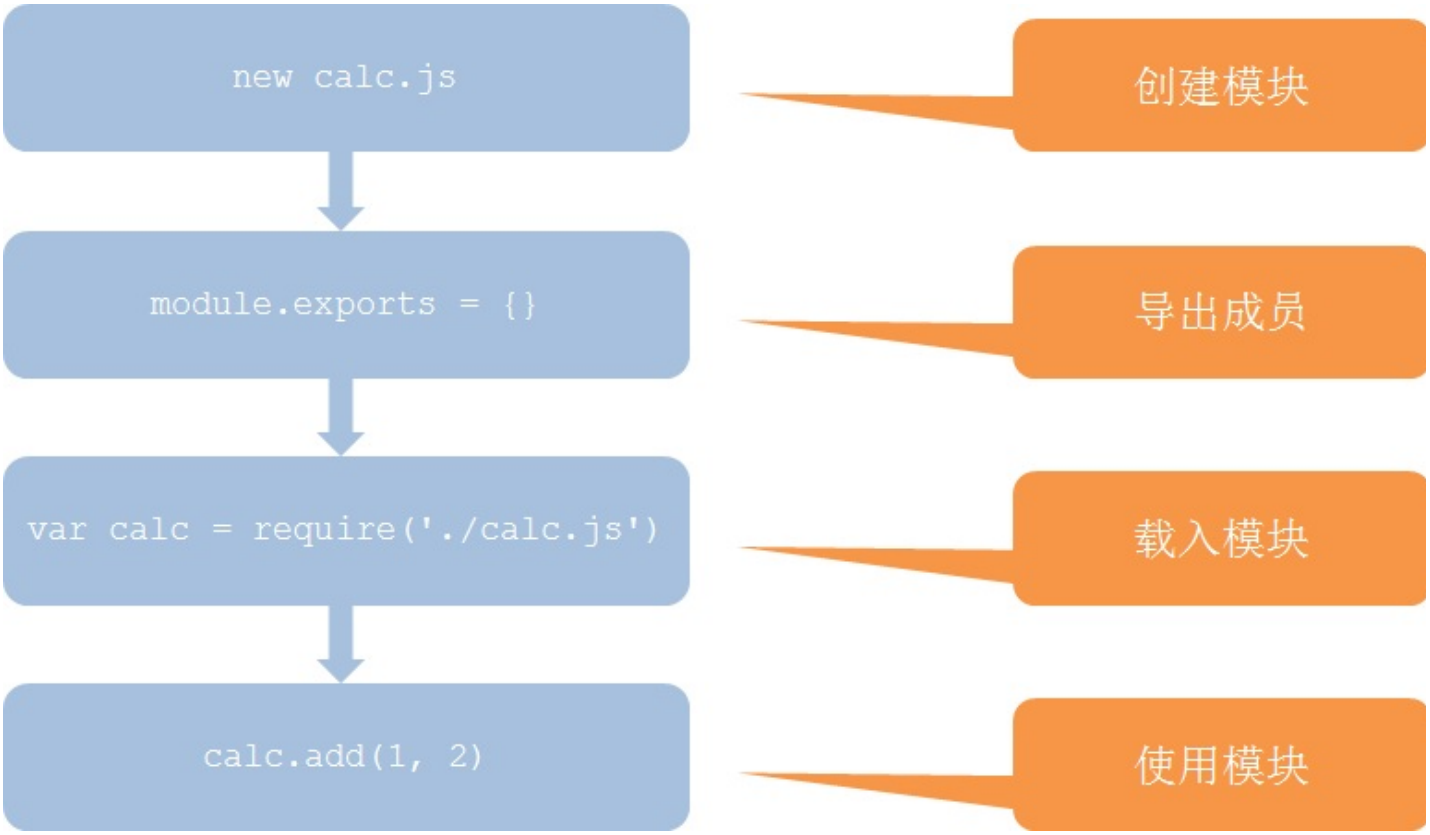
> CommonJS 就是一套约定标准，不是技术；用于约定我们的代码应该是怎样的一种结构；
> http://wiki.commonjs.org/wiki/CommonJS

> CommonJS 模块的特点
> - 所有代码都运行在模块作用域，不会污染全局作用域。
> - 模块可以多次加载，但是只会在第一次加载时运行一次，然后 运行结果就被缓存了，以后再加载，就直接读取缓存结果。要想让模块再次运行，必须清除缓存。
> - 模块加载的顺序，按照其在代码中出现的顺序。

##模块的分类

- 文件模块
  就是我们自己写的功能模块文件
- 核心模块
  Node 平台自带的一套基本的功能模块，也有人称之为 Node 平台的 API
- 第三方模块
  社区或第三方个人开发好的功能模块，可以直接拿回来用

##模块化开发的流程



##模块内全局环境（伪）

- 我们在之后的文件操作中必须使用绝对路径
- __dirname

  用于获取当前文件所在目录的完整路径；

  在 REPL 环境无效；
- __filename

  用来获取当前文件的完整路径；

  在 REPL 环境同样无效；
- module

  模块对象
- exports

  映射到 module.exports 的别名
- require()

  require.cache

  require.extensions

  require.main

  require.resolve()

###module 对象

- Node 内部提供一个 Module 构建函数。所有模块都是 Module 的实例，属性如下：

module.id 模块的识别符，通常是带有绝对路径的模块文件名。

module.filename 模块定义的文件的绝对路径。

module.loaded 返回一个布尔值，表示模块是否已经完成加载。

module.parent 返回一个对象，表示调用该模块的模块。

module.children 返回一个数组，表示该模块要用到的其他模块。

module.exports 表示模块对外输出的值。

- 载入一个模块就是构建一个 Module 实例。

##模块的定义

- 一个新的 JS 文件就是一个模块；
- 一个合格的模块应该是有导出成员的，否则模块就失去了定义的价值；
- 模块内部是一个独立（封闭）的作用域（模块与模块之间不会冲突）；
- 模块之间必须通过导出或导入的方式协同；
- 导出方式：

```
exports.name = value;
module.exports = { name: value };
```

module.exports 和 exports

- module.exports 是用于为模块导出成员的接口
- exports 是指向 module.exports的别名，相当于在模块开始的时候执行：`var exports = module.exports`；
- 一旦为 module.exports赋值，就会切断之前两者的相关性；
- 最终模块的导出成员以 module.exports 为准

##载入模块

require 函数

###require 简介

- Node 使用 CommonJS 模块规范，内置的 require 函数用于加载模块文件。
- require 的基本功能是，读入并执行一个 JavaScript 文件，然后返回该模块的 exports 对象。
- 如果没有发现指定模块，会报错。

##require 扩展名

```
require 加载文件时可以省略扩展名：
require('./module');
// 此时文件按 JS 文件执行
```

```
require('./module.js');
// 此时文件按 JSON 文件解析
require('./module.json');
// 此时文件预编译好的 C++ 模块执行
require('./module.node');
// 载入目录module目录中的 package.json 中main指向的文件
require('./module/default.js');
// 载入目录module 中的index.js文件
```

##require 加载文件规则

- 通过 ./ 或 ../ 开头：则按照相对路径从当前文件所在文件夹开始寻找模块；
  require('../file.js'); => 上级目录下找 file.js 文件
- 通过 / 开头：则以系统根目录开始寻找模块；require('/Users/iceStone/Documents/file.js'); => 以绝对路径的方式找
  没有任何异议
- 如果参数字符串不以 “./ “ 或 ” / “ 开头，则表示加载的是一个默认提供的核心模块（位于 Node 的系统安装目录中）：
  require('fs'); => 加载核心模块中的文件系统模块
  或者从当前目录向上搜索 node_modules 目录中的文件：require('my_module'); => 各级 node_modules 文件夹中搜索 my_module.js 文件；

###require 加载目录规则

- 如果 require 传入的是一个目录的路径，会自动查看该目录的 package.json 文件，然后加载 main 字段指定的入口文件
- 如果package.json文件没有main字段，或者根本就没有package.json文件，则默认找目录下的 index.js 文件作为模块：
  require('./calcuator'); => 当前目录下找 calculator 目录中的 index.js 文件

##常用内置模块清单

如果只是在服务器运行 JavaScript代码，意义并不大，因为无法实现任何功能（读写文件，访问网络）。
Node 的用处在于它本身还提供的一系列功能模块，用于与操作系统互动。
这些核心的功能模块在 Node 中内置。
内置如下模块：

- path：处理文件路径。
- fs：操作文件系统。
- child_process：新建子进程。
- util：提供一系列实用小工具。

- http：提供HTTP服务器功能。
- url：用于解析URL。
- querystring：解析URL中的查询字符串。
- crypto：提供加密和解密功能。
- 其他

实现require和cache

##require 的实现机制

- 将传入的模块 ID 通过加载规则找到对应的模块文件
- 读取这个文件里面的代码
- 通过拼接的方式为该段代码构建私有空间
- 执行该代码
- 拿到 module.exports 返回

##模块的缓存

- 第一次加载某个模块时，Node 会缓存该模块。以后再加载该模块，就直接从缓存取出该模块的 module.exports 属性（不会再次执行该模块）
- 如果需要多次执行模块中的代码，一般可以让模块暴露行为（函数）
- 模块的缓存可以通过 require.cache 拿到，同样也可以删除

index.js

```
// 模块的缓存

'use strict';

function $require(id) {
  const fs = require('fs');
  const path = require('path');

  const filename = path.join(__dirname, id); // pathto/module1.js
  $require.cache = $require.cache || {};
  if ($require.cache[filename]) {
    //
    return $require.cache[filename].exports;
  }

  // 没有缓存 第一次
  const dirname = path.dirname(filename);  // pathto

  let code = fs.readFileSync(filename, 'utf8');

  // 定义一个数据容器，用容器去装模块导出的成员
  let module = { id: filename, exports: {} };
  let exports = module.exports; // module.exports

  code = `
```

```
  (function($require, module, exports, __dirname, __filename) {
    ${code}
  })($require, module, exports, dirname, filename);`;
  eval(code);

  // 缓存起来
  $require.cache[filename] = module;

  return module.exports;
}

setInterval(() => {
  var date = $require('./module/date.js');
  console.log(date.getTime());
}, 1000);
```

module/date.js

```
console.log('date module exec');
module.exports = new Date();
```

核心模块操作

这些核心的功能模块在 Node 中内置。

内置如下模块：

- path：处理文件路径。
- fs：操作文件系统。
- child_process：新建子进程。
- util：提供一系列实用小工具。
- http：提供HTTP服务器功能。
- url：用于解析URL。
- querystring：解析URL中的查询字符串。
- crypto：提供加密和解密功能。
- 其他

fs文件系统操作

文件操作

相关模块

Node内核提供了很多与文件操作相关的模块，每个模块都提供了一些最基本的操作API，在NPM中也有社区提供的功能包

fs：

基础的文件操作 API

path：

提供和路径相关的操作 API（在文件操作的过程中，都”必须”使用物理路径（绝对路径））

readline：

用于读取大文本文件，一行一行读

fs-extra（第三方）：

https://www.npmjs.com/package/fs-extra

同步调用和异步调用

##同步或异步调用

- fs模块对文件的几乎所有操作都有同步和异步两种形式
- 例如：readFile() 和 readFileSync()
- 区别：
  - 同步调用会阻塞代码的执行，异步则不会
  - 异步调用会将读取任务下达到任务队列，直到任务执行完成才会回调
  - 异常处理方面，同步必须使用 try catch 方式，异步可以通过回调函数的第一个参数

例子：

1.读文件写法不一样（eadFile() 和 readFileSync()）

2.捕捉错误方式不一样

```
// 同步调用和异步调用

const fs = require('fs');
const path = require('path');

//同步调用（同步调用的方式可以使用try catch方式，阻塞读取完）
console.time('sync');
try {
  var data = fs.readFileSync(path.join('/tmp', 'test.js'));
  // console.log(data);
} catch (error) {
  throw error;
}
console.timeEnd('sync');

//异步调用
console.time('async');
fs.readFile(path.join('/tmp', 'test.js'), function(error, data)  {
  if (error) throw error;
  // console.log(data);
});
console.timeEnd('async');
```

结果：

```
sync: 11ms
async: 1ms
```

缓冲区处理（二进制数据）

> https://0532.gitbooks.io/nodejs/content/buffers/README.html

> buffer API https://nodejs.org/api/buffer.html

> 参考：http://blog.fens.me/nodejs-buffer/

##什么是缓冲区

- 缓冲区就是内存中操作数据的容器
- 只是数据容器而已
- 通过缓冲区可以很方便的操作二进制数据
- 而且在大文件操作时必须有缓冲区

##为什么要有缓冲区

- JS 是比较擅长处理字符串，但是早期的应用场景主要用于处理 HTML 文档，不会有太大篇幅的数据处理，也不会接触到二进制的数据。
- 而在 Node 中操作数据、网络通信是没办法完全以字符串的方式操作的，简单来说
- 所以在 Node 中引入了一个二进制的缓冲区的实现：Buffer

##缓冲区操作

- 创建长度为4个字节的缓冲区

```
var buffer = new Buffer(4);
```

- 通过指定数组内容的方式创建

```
var buffer = new Buffer([00,01]);
```

- 通过指定编码的方式创建

```
var buffer = new Buffer('hello', 'utf8');
```

##例子

buf.write(string[, offset[, length]][, encoding])

write写入的是字符串

有offset偏移，如果不指定，每次写，都是从头写。

.toString('utf8')

以文本方式读

其他：

大端、小端的区别(BIG-ENDIAN OR LITTLE-ENDIAN)

- buf.write(string[, offset[, length]][, encoding])
- buf.writeDoubleBE(value, offset[, noAssert])
- buf.writeDoubleLE(value, offset[, noAssert])
- buf.writeFloatBE(value, offset[, noAssert])
- buf.writeFloatLE(value, offset[, noAssert])
- buf.writeInt8(value, offset[, noAssert])
- buf.writeInt16BE(value, offset[, noAssert])
- buf.writeInt16LE(value, offset[, noAssert])
- buf.writeInt32BE(value, offset[, noAssert])
- buf.writeInt32LE(value, offset[, noAssert])
- buf.writeIntBE(value, offset, byteLength[, noAssert])
- buf.writeIntLE(value, offset, byteLength[, noAssert])
- buf.writeUInt8(value, offset[, noAssert])
- buf.writeUInt16BE(value, offset[, noAssert])
- buf.writeUInt16LE(value, offset[, noAssert])
- buf.writeUInt32BE(value, offset[, noAssert])
- buf.writeUInt32LE(value, offset[, noAssert])
- buf.writeUIntBE(value, offset, byteLength[, noAssert])
- buf.writeUIntLE(value, offset, byteLength[, noAssert])

例子1：

```
// 读取文件
const fs = require('fs');
const path = require('path');


// 读取文件时没有指定编码默认读取的是一个Buffer(缓冲区)
// readFile的方式确实是使用buffer,但是也是一次性读取
fs.readFile(path.join(__dirname, './../README.md'), (error, data) => {
   console.log(data.toString('utf8'));
});
```

例子2：

```
var buffer = new Buffer(4);

buffer.write('sssssssss');

console.log(buffer.toString('utf8'));
```

结果

```
ssss
```

例子3：

```
var buffer = new Buffer(4);

buffer.write('12');

console.log(buffer.toString('utf8'));
console.log(buffer.toString('utf8', 0, 2));
console.log(buffer.toString('utf8', 0, 1));

//offset
buffer.write('77');
buffer.write('88', 2);
console.log(buffer.toString('utf8'));
```

结果

进行了补位

```
12
12
1
7788
```

例子4：

init32和int16的值范围

```
// -272323123 <= int32 < 27120832109
// -272323123 <= int16 < 27120832109
```

##应用（读取图片，转成bash64）

```
// 读取图片

const fs = require('fs');
const path = require('path');

fs.readFile(path.join(__dirname, './../img/google.png'), (error, data) => {
   console.log(data.toString('base64'));
 });
```

DataURI详解

```
data:image.png;bash64,xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

##Node 默认支持的编码

Buffers 和 JavaScript 字符串对象之间转换时需要一个明确的编码方法。下面是字符串的不同编码。

- 'ascii' - 7位的 ASCII 数据。这种编码方式非常快，它会移除最高位内容。
- 'utf8' - 多字节编码 Unicode 字符。大部分网页和文档使用这类编码方式。
- 'utf16le' - 2个或4个字节, Little Endian (LE) 编码 Unicode 字符。编码范围 (U+10000 到 U+10FFFF) 。
- 'ucs2' - 'utf16le'的子集。
- 'base64' - Base64 字符编码。
- 'binary' - 仅使用每个字符的头8位将原始的二进制信息进行编码。在需使用 Buffer 的情况下，应该尽量避免使用这个已经过时的编码方式。这个编码方式将会在未来某个版本中弃用。
- 'hex' - 每个字节都采用 2 进制编码。

默认不支持GBK

使用 iconv-lite插件 提供编码转换

> https://www.npmjs.com/package/iconv-lite

Node.js 处理 GBK 编码模板

例子

```
// 解决GBK编码支持问题
```

```
// 所有的文件操作全部基于FS模块
const fs = require('fs');
// 无论是同步操作还是异步操作都“必须使用”绝对路径的形式操作
const path = require('path');

// 将文本读取到一个buffer中
const buffer = fs.readFileSync(path.join(__dirname, '../lyrics/友谊之光.lrc'));

// 由于Windows下文件默认编码为GBK所以需要通过
const iconv = require('iconv-lite');
const content2 = iconv.decode(buffer,'gbk');
console.log(content2);
```

文件读取

##文件读取

- 异步文件读取

```
fs.readFile(file[,options],callback(err,data))
```

- 同步文件读取

```
fs.readFileSync(file,[,option])
```

- 文件流的方式读取（后面单独介绍）

```
fs.createReadStream(path[, options])
```

基本的读取文件

```
// 所有的文件操作全部基于FS模块
const fs = require('fs');
// 无论是同步操作还是异步操作都“必须使用”绝对路径的形式操作
const path = require('path');

// 同步的方式读取一个文本文件
try {
  const content = fs.readFileSync(path.join(__dirname, '../lyrics/友谊之光.lrc'))
;
  console.log(content);
} catch (error) {
  th
```

读取文本文件内容

```
// 所有的文件操作全部基于FS模块
const fs = require('fs');
// 无论是同步操作还是异步操作都“必须使用”绝对路径的形式操作
const path = require('path');

// 将文本读取到一个buffer中
const buffer = fs.readFileSync(path.join(__dirname, '../lyrics/友谊之光.lrc'));

// 将buffer中的内容读取出来
```

```
const content = buffer.toString();
console.log(content);
```

##readline 模块逐行读取文本

```
const readline = require('readline');
const fs = require('fs');

const rl = readline.createInterface({
  input: fs.createReadStream('sample.txt')
});

rl.on('line', function(line) {
  console.log('Line from file:', line);
});
```

文件写入

##文件写入

- 确保操作没有额外的问题，一定使用绝对路径的方式
- 异步文件写入

```
fs.writeFile(file,data[,option],callback(err))
```

- 同步文件写入

```
fs.writeFileSync(file,data,[,option])
```

- 流式文件写

```
fs.createWriteStream(path[,option])
```

默认写入操作是覆盖源文件

- 异步追加

```
fs.appendFile(file,data[,options],callback(err))
```

- 同步追加

```
fs.appendFileSync(file,data[,options])
```

```
// 文件写入

const fs = require('fs');
const path = require('path');

//同步文件写入
try {
  fs.writeFileSync(path.join(__dirname, '../lyrics/temp.txt'), new Date());
} catch (error) {
  // 文件夹不存在，或者权限错误
  console.log(error);
}

//异步文件写入
```

```
fs.writeFile(path.join(__dirname, '../lyrics/temp.txt'), new Date(), function(error) {
  console.log(error);
});

//流式文件写
var streamWriter = fs.createWriteStream(path.join(__dirname, '../lyrics/temp.txt'));

setInterval(() => {
  streamWriter.write(`${new Date}\n`, function(error) {
    console.log(error);
  });
}, 1000);

//异步追加
setInterval(() => {
  fs.appendFile(path.join(__dirname, '../lyrics/temp.txt'),`${new Date}\n`, function(error) {
     console.log(error);
   });
 }, 1000);

//同步追加
setInterval(() => {
  fs.appendFileSync(path.join(__dirname, '../lyrics/temp.txt'),`${new Date}\n`);

}, 1000);
```

例子：读取歌词文件显示

##例子：读取歌词文件显示

JavaScript RegExp 对象

歌词文件:传奇.lrc

```
 [ti:传奇]
[ar:李健]
[al:李春天的春天]
[offset:0]
[00:01.50]传奇 - -李键
[00:04.63]词：左右  曲：李键
[00:08.81]
[00:35.45]是只因为在人群中多看了你一眼
[00:42.95]再也没能忘掉你容颜
[00:50.54]梦想着偶然能有一天再相见
[00:57.77]从此我开始孤单思念
[01:04.75]想你时你在天边
[01:11.98]想你时你在眼前
[01:19.72]想你时你在脑海
[01:27.02]想你时你在心田
[01:35.47]宁愿相信我们前世有约
[01:41.76]今生的爱情故事不会再改变
[01:50.37]宁愿用这一生等你发现
[01:56.80]我一直在你身旁
[01:59.48]从未走远
[02:38.17]只是因为在人群中多看了你一眼
[02:45.53]再也没能忘掉你容颜
[02:53.08]梦想着偶然能有一天再相见
[03:00.58]从此我开始孤单思念
[03:07.57]想你时你在天边
[03:14.92]想你时你在眼前
[03:22.55]想你时你在脑海
[03:29.91]想你时你在心田
[03:38.13]宁愿相信我们前世有约
[03:44.25]今生的爱情故事不会再改变
[03:53.00]宁愿用这一生等你发现
[03:59.44]我一直在你身旁
[04:01.94]从未走远
[04:08.30]宁愿相信我们前世有约
[04:14.41]今生的爱情故事不会再改变
[04:23.27]宁愿用这一生等你发现
[04:29.39]我一直在你身旁
[04:31.95]从未走远
[04:38.16]只是因为在人群中多看了你一眼
[04:47.01]
```

```
歌词来源:www.lrcxz.com
```

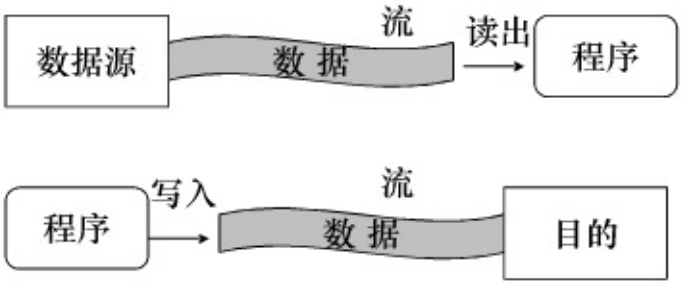
index.js

```
// readline实现一行一行读取歌词

const fs = require('fs');
const path = require('path');
const iconv = require('iconv-lite');
const readline = require('readline');

var readStream = fs.createReadStream(path.join(__dirname, './传奇.lrc')).pipe(i
conv.decodeStream('gbk'));
var rl = readline.createInterface({ input: readStream });

var regex = /\[(\d{2}):(\d{2})\.(\d+)\](.+)/;
rl.on('line', function(line) {
  var time = regex.exec(line);
  if (time) {
    var m = parseInt(time[1]);
    var s = parseInt(time[2]);
    var ms = parseInt(time[3]);
    var all = m * 60 * 1000 + s * 1000 + ms;
    setTimeout(function() {
      console.log(time[4]);
    }, all);
  } else {
    console.log(line);
  }
});
;
```

文件流

##文件流

- 文件流就是以面向对象的概念对文件数据进行的抽象
- 文件流定义了一些对文件数据的操作方式



- 流式文件写

```
fs.createWriteStream(path[,option])
```

- 流式文件读

```
fs.createReadStream(path[, options])
```

##例子

```
// 文件流的方式读取文件内容

const fs = require('fs');
const path = require('path');
const iconv = require('iconv-lite');

// 创建一个文件读取流
var stream = fs.createReadStream(path.join(__dirname, './../lyrics/血染的风采.lr
c'));
// 让文件流通过iconv过滤编码
stream = stream.pipe(iconv.decodeStream('gbk'));
// 流到输出控制台
// stream.pipe(process.stdout);

var data = '';
stream.on('data', function(trunk) {
  console.log(trunk);
```

```
});
stream.on('end', function() {
  console.log('end');
});
```

例子：文件复制

##文件的复制

###非流异步方式

```
const fs = require('fs');
const path = require('path');

console.time('read');
fs.readFile('/tmp/1.iso', function(data, err) {
  if (err) {
    throw err
  }
  console.timeEnd('read');
  console.time('write');
  // 读取完文件拿到
  fs.writeFile('/tmp/2.iso', function(data, err) {
    if (err) {
      throw err
    }
    console.timeEnd('write');
    console.log('拷贝完成');
  });
});
```

缺陷：

- 大文件拷贝，内存受不了
- 没有进度的概念

文件流的方式的复制

```
const fs = require('fs');
const path = require('path');

// 创建文件的读取流，并没有读出正式的数据，开始了读取文件的任务（）
var reader = fs.createReadStream('/tmp/1.iso');
// 创建一个写入流
var writer = fs.createWriteStream('/tmp/1.iso');

// 磁盘： 7200转 6100转 转速越快 读写越快 资源消耗更大
fs.stat('/tmp/1.iso', function (err, stats) {
  if (stats) {
    var readTotal = 0;
```

```
    reader.on('data', function(chunk) {
      // chunk是一个buffer(字节数组)
      writer.write(chunk, function (err) {
        console.log('写 进度：' + ((readTotal += chunk.length) / stats.size * 100
) + '%');
      });
    });

    reader.on('end', function() {
      // 没有了
    });
  }
});
```

##使用pipe

看api文档

监视文件

##监视文件

监视文件变化：

- fs.watchFile(filename[, options], listener(curr,prev))
  - options:{persistent,interval}
- fs.watch(filename[,options][,listener])

##例子：

利用文件监视实现自动 markdown 文件转换

https://github.com/chjj/marked

https://github.com/Browsersync/browser-sync

##Markdown文件自动转换

- 实现思路：
  1. 利用 `fs` 模块的文件监视功能监视指定MD文件
  2. 当文件发生变化后，借助 `marked` 包提供的 `markdown` to `html` 功能将改变后的MD文件转换为HTML
  3. 再将得到的HTML替换到模版中
  4. 最后利用BrowserSync模块实现浏览器自动刷新

     browsersync需要用到Python

```
const fs = require('fs');
const path = require('path');
const marked = require('marked');
const browserSync = require("browser-sync");

// 接收需要转换的文件路径

const target = path.join(__dirname, process.argv[2] || '../README.md');

// 转换为HTML后保存的位置
var filename = target.replace(path.extname(target), '.html');

// 获取HTML文件名
var indexpath = path.basename(filename);


// 通过browsersync创建一个文件服务器
browserSync({
  notify: false,
```

```
  server: path.dirname(target), // 网站根目录
  index: indexpath // 默认文档：（如果浏览器访问一个目录的话，默认返回那个文件）
});

// 监视文件变化
fs.watchFile(target, { interval: 200 }, (curr, prev) => {
  // 一旦文件变化，触发该函数

  // 判断文件到底有没有变化，  减少不必要的转换
  if (curr.mtime === prev.mtime) {
    return false;
  }

  // 读取文件 转换为新的HTML
  fs.readFile(target, 'utf8', (err, content) => {
    if (err) {
      throw err;
    }
    var html = marked(content);

    // 注入CSS样式
    fs.readFile(path.join(__dirname, 'github.css'), 'utf8', (err, css) => {
      html = template.replace('{{{content}}}', html).replace('{{{styles}}}', css
);
      // 这里的HTML就已经有内容 有样式

      fs.writeFile(filename, html, 'utf8', (err) => {
        // 通过browserSync发送一个消息给浏览器，流量器刷新
        browserSync.reload(indexpath);
        console.log('updated@' + new Date);
      });
    });
  });
});

var template = `
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <style>{{{styles}}}</style>
</head>
<body>
  <div class="vs">
    {{{content}}}
  </div>
</body>
</html>
`;
```

```
*****
```

其他文件操作

##其他文件操作

- 验证路径是否存在（过时的API）

```
fs.exists(path,callback(exists))
fs.existsSync(path) // => 返回布尔类型 exists
```

- 获取文件信息

```
fs.stat(path,callback(err,stats))
fs.statSync(path) // => 返回一个fs.Stats实例
```

- 移动文件

```
fs.rename(oldPath,newPath)
```

- 重命名文件或目录

```
fs.rename(oldPath,newPath,callback)
fs.renameSync(oldPath,newPath)
```

删除文件

```
fs.unlink(path,callback(err))
fs.unlinkSync(path)
```

##例子

移动文件和重名

```
const fs = require('fs');
const path = require('path');


var currentPath = path.join(__dirname, '../temp1.txt');
var targetPath = path.join(__dirname, '../img/temp1.txt');

fs.rename(currentPath, targetPath);
```

目录操作

##目录操作

- 创建一个目录

```
fs.mkdir(path[,model],callback)
fs.mkdirSync(path[,model])
```

- 删除一个空目录

```
fs.rmdir(path,callback)
fs.rmdirSync(path)
```

- 读取一个目录

```
fs.readdir(path,callback(err,files))
fs.readdirSync(path) // => 返回files
```

##例子

打印当前目录所有文件

```
// 打印当前目录所有文件

const fs = require('fs');
const path = require('path');
require('./proto.js');

// 获取当前有没有传入目标路径
var target = path.join(__dirname, process.argv[2] || './');



fs.readdir(target, (err, files) => {
  files.forEach(file=> {
    console.time(file);
    // console.log(path.join(target, file));
    fs.stat(path.join(target, file), function(err, stats) {
      // stats.mtime.
      console.log(`${stats.mtime.format('yyyy/MM/dd HH:mm') }\t${stats.size}\t${
file}`);
      console.timeEnd(file);
    });
```

```
  });
});
```

proto.js

```
/*
* @Author: iceStone
* @Date:   2016-01-07 22:28:04
* @Last Modified by:   iceStone
* @Last Modified time: 2016-01-07 22:28:12
*/

'use strict';

// 对Date的扩展，将 Date 转化为指定格式的String
// 月(M)、日(d)、小时(h)、分(m)、秒(s)、季度(q) 可以用 1-2 个占位符，
// 年(y)可以用 1-4 个占位符，毫秒(S)只能用 1 个占位符(是 1-3 位的数字)
// 例子：
// (new Date()).Format("yyyy-MM-dd hh:mm:ss.S") ==> 2006-07-02 08:09:04.423
// (new Date()).Format("yyyy-M-d h:m:s.S")      ==> 2006-7-2 8:9:4.18
Date.prototype.format = function(format) { //author: meizz
  let o = {
    "M+": this.getMonth() + 1, //月份
    "d+": this.getDate(), //日
    "H+": this.getHours(), //小时
    "m+": this.getMinutes(), //分
    "s+": this.getSeconds(), //秒
    "q+": Math.floor((this.getMonth() + 3) / 3), //季度
    "f+": this.getMilliseconds() //毫秒
  };
  if (/(y+)/.test(format))
    format = format.replace(RegExp.$1, (this.getFullYear() + "").substr(4 - Reg
Exp.$1.length));
  for (let k in o)
    if (new RegExp("(" + k + ")").test(format))
      format = format.replace(RegExp.$1, (RegExp.$1.length == 1) ? (o[k]) : (("
00" + o[k]).substr(("" + o[k]).length)));
  return format;
};
```

创建目录

```
// 创建文件夹
//创建目录的上级目录必须存在，否则创建失败

const fs = require('fs');
```

```
const path = require('path');

//成功
fs.mkdir(path.join(__dirname, 'demo1'));


//报错，上级目录demo2不存在
 fs.mkdir(path.join(__dirname, 'demo2/demo3'), (err) => {
   console.log(err);
 });
```

创建层级目录

以模块的方式

步骤：

创建模块文件，定义模块成员，导出模块成员，载入模块，使用模块

module/mkdirs.js

```
// 创建层级目录

const fs = require('fs');
const path = require('path');

function mkdirs(pathname, callback) {
  // module.parent 拿到的是调用我的对象 index.js
  // console.log(module.parent);
  var root = path.dirname(module.parent.filename);

  pathname = path.isAbsolute(pathname) ? pathname : path.join(root, pathname)

  var relativepath = path.relative(root, pathname);

  var folders = relativepath.split(path.sep);

  try {
    var pre = '';
    folders.forEach(folder => {
      try {
        // 如果不存在则读取不到文件内容则报错
        fs.statSync(path.join(root, pre, folder));
      } catch (error) {
        fs.mkdirSync(path.join(root, pre, folder));
      }

      pre = path.join(pre, folder);
    });
    callback && callback(null);
  } catch (error) {
```

```
    callback && callback(error);
  }
}

module.exports = mkdirs;
```

index.js

```
var mkdirs = require('./module/mkdirs');
var path = require('path');

mkdirs(path.join(__dirname, 'f1/f2/f3'), (err) => { console.log(err); });
```

注意__dirname，是运用文件所在的目录。

例子：递归加载目录树

##递归加载目录树

```
// 打印当前目录树

const fs = require('fs');
const path = require('path');

var target = path.join(__dirname, process.argv[2] || '../');
console.log(target);


function loaddir(target, level) {
  var prefix = new Array(level + 1).join('|  ');
  var dirinfo = fs.readdirSync(target);

  var dirs = [];

  var files = [];

  dirinfo.forEach(info=> {
    var stat = fs.statSync(path.join(target, info));
    if (stat.isDirectory()) {
      dirs.push(info);
    } else {
      files.push(info);
    }
  });

  var next = level + 1;
  dirs.forEach(dir=> {
    console.log(`${prefix}├─ ${dir}`);
    loaddir(path.join(target, dir), next);
  });

  var count = files.length - 1;
  files.forEach(file=> {
    if (count--) {
      console.log(`${prefix}├─ ${file}`);
    } else {
      console.log(`${prefix}└─ ${file}`);
    }
  });
}

loaddir(target, 0);
```

```
// function loaddir(target, level) {
//   var line = new Array(level).join(' ');
//   var dirinfo = fs.readdirSync(target);

//   var dirs = [];
//   var files = [];

//   dirinfo.forEach(info=> {
//     var stat = fs.statSync(path.join(target, info));
//     if (stat.isDirectory()) {
//       dirs.push(info);
//     } else {
//       files.push(info);
//     }
//   });

//   dirs.forEach(dir=> {
//     console.log(`${line}└─ ${dir}`);
//     loaddir(path.join(target, dir), 1 + level);
//   });

//   var count = files.length - 1;
//   files.forEach(file=> {
//     if (count--) {
//       console.log(`${line}├─ ${file}`);
//     } else {
//       console.log(`${line}└─ ${file}`);
//     }
//   });
// }

// loaddir(target, 0);
```

path路径操作模块

##path

- 在文件操作的过程中，都”必须”使用物理路径（绝对路径）
- path 模块提供了一系列与路径相关的 API
  - path.join([p1][,p2][,p3]...) => 连接多个路径
  - path.basename(p, ext) => 获取文件名
  - path.dirname(p) => 获取文件夹路径
  - path.extname(p) => 获取文件扩展名
  - path.format(obj) 和 path.parse(p)
  - path.relative(from, to) => 获取从 from 到 to 的相对路径
- 源码地址：
  https://github.com/nodejs/node/blob/master/lib/path.js
- API地址
  http://nodejs.org/api/path.html

##path模块使用

路径操作，非文件操作，所以无需path路径是否存在

```
// PATH模块的使用

const path = require('path');

const temp = path.join(__dirname, './../测试文件.txt');
```

- path.basename(p[, ext])

```
//语法： path.basename(p[, ext])

// 获取文件名
console.log(path.basename(temp));

// 获取文件名without扩展名
 console.log(path.basename(temp, 'txt'));
```

结果：

```
测试文件.txt
测试文件
```

- path.delimiter

```
//语法： path.delimiter
// 获取不同操作系统中默认的路径分隔符 Windows是; Linux是:
console.log(path.delimiter);
// 获取环境变量
 console.log(process.env.PATH.split(path.delimiter));
```

结果：

```
 :
[ '/usr/lib/jvm/java/bin',
  '/usr/lib/jvm/java/bin',
  '/usr/local/sbin',
  '/usr/local/bin',
  '/usr/sbin',
  '/usr/bin',
  '/sbin',
  '/bin',
  '/usr/games',
  '/usr/local/games' ]
```

- path.dirname(p)

```
//语法： path.dirname(p)

// 获取文件目录名称
 console.log(path.dirname(temp));
```

- path.extname(p)

```
// 语法：path.extname(p)
// 获取指定文件的扩展名，包含.
 console.log(path.extname(temp));
```

结果

```
 .txt
```

- path.parse(pathString)

```
//语法： path.parse(pathString)

// 将一个路径字符串转换为一个对象（包含文件目录，文件名，扩展名）
 var obj = path.parse(temp);
 console.log(obj);
```

结果：

```
{ root: '/',
  dir: '/media/revin/WININSTALL/3/新建文件夹',
  base: '测试文件.txt',
  ext: '.txt',
  name: '测试文件' }
```

- path.format(pathObject)

```
//语法： path.format(pathObject)

// 将路径对象转字符串
var obj = path.parse(temp);
console.log(path.format(obj));
```

结果：

```
/media/revin/WININSTALL/3/新建文件夹/测试文件.txt
```

- path.isAbsolute(path)

```
// 语法： path.isAbsolute(path)
//判断路径是否是绝对路径
// // true
 console.log(path.isAbsolute(temp));
// // false
 console.log(path.isAbsolute('../temp/1.txt'));
```

- path.join([path1][, path2][, ...])

```
// 语法： path.join([path1][, path2][, ...])
// 拼合路径组成
console.log(path.join(__dirname, '..', './temp', 'a', '../../1.txt'));
console.log(path.join('/foo', 'bar', 'baz/asdf', 'quux', '..'));
```

结果：

```
/media/revin/WININSTALL/3/新建文件夹/1.txt
/foo/bar/baz/asdf
```

- path.normalize(p)

```
//语法： path.normalize(p)
// 常规化一个路径（为windows设计）
 var a = path.normalize('C:/dev\\abc//cba////1.txt');
 console.log(a);
```

结果：

```
 C:/dev\abc/cba/1.txt
```

- path.relative(from, to)

```
 // 语法：path.relative(from, to)
  console.log(path.relative(__dirname,'/media/revin/WININSTALL/3/新建文件夹/code
/0122.js'));
// 获取to 相对于from的相对路径
```

结果：

因为

当前路径是/media/revin/WININSTALL/3/新建文件夹/code/,to 相对于from的相对路径是

```
0122.js
```

- path.resolve([from ...], to)

```
//语法： path.resolve([from ...], to)
//
 console.log(path.resolve(__dirname, '..', './', './code'));
// 与join不同（resolve会执行cd /tmp到目录）
  console.log(path.resolve(__dirname, '/tmp', './', './code'));
 console.log(path.join(__dirname, '/tmp', './', './code'));
```

结果：

```
/media/revin/WININSTALL/3/新建文件夹/code
```

```
/tmp/code
/media/revin/WININSTALL/3/新建文件夹/code/tmp/code
```

- path.sep

```
// path.sep
// 获取当前操作系统中默认用的路径成员分隔符 windows:\ linux:/
 console.log(path.sep);
```

结果：

```
/
```

- path.win32
- path.posix

```
// 根据操作系统决定

//path.win32
// 允许在任意操作系统上使用windows的方式操作路径
path === path.win32


// 允许在任意操作系统上使用Linux的方式操作路径
// path.posix

path === path.posix
```

网络操作

##网络操作

- url

  用于解析 URL 格式的模块
- querystring

  用于操作类似 k1=v1&k2=v2 的查询字符串
- http

  用于创建 HTTP 服务器或 HTTP 客户端

URL 解析模块

##URL 解析模块

- 将一个 URL 字符串解析为一个 URL 对象

```
url.parse(urlStr[, parseQueryString][, slashesDenoteHost])
```

- 将一个 URL 对象格式化为字符串的形式

```
url.format(urlObj)
```

用于组合 URL 成员为完整的 URL 字符串

```
url.resolve(from, to)
```

querystring查询字符串模块

##querystring查询字符串模块

- querystring.escape
- querystring.parse(str[, sep][, eq][, options])
- querystring.stringify(obj[, sep][, eq][, options])
- querystring.unescape

crypto加密解密模块

##crypto加密解密模块

Socket

##基本操作例子

server.js

```
// 建立一个Socket服务端
const net = require('net');
// 创建一个Socket服务器
var server = net.createServer(function socketConnect(socket) {
  // 当有客户端与我连接的时候出发
  // console.log(`${socket.remoteAddress}:${socket.remotePort} 进来了`);
  // socket.write(`hello ${socket.remoteAddress}:${socket.remotePort} 你来了`)

  // 监听socket有数据过来
  socket.on('data', function(chunk)  {
    console.log(chunk.toString());
    socket.write('server > 你说啥？');
  });
});

var port = 2080;
// 监听特定的端口
server.listen(port, function(err)  {
  // 成功监听 2080 端口过后执行 如果监听失败（端口被别人用了）会有ERROR
  if (err) {
    console.log('端口被占用');
    return false;
  }
  console.log('服务端正常启动监听【${port}】端口');
})
```

client.js

```
// 建立socket客户端
const net = require('net');
const socket = net.connect({ port: 2080 }, function() {
  this.debug(arguments);
},function() {
  console.log('已经连接到服务端!');


  process.stdout.write('\nclient > ');
  process.stdin.on('data', function(chunk)  {
    // 控制台输入回车
    // console.log(chunk.toString().trim());
    socket.write(chunk.toString().trim());
```

```
    process.stdout.write('\nclient >');
  });

  socket.on('data', function(data)  {
    console.log('\n' + data.toString());
  });
});

 socket.on('end', function()  {
  console.log('disconnected from server');
});
```

例子：聊天室

##聊天室例子

提示：readline promat()解释

http://blog.csdn.net/zgljl2012/article/details/48321171

server.js

```
// 建立一个Socket服务端

const net = require('net');

// 用于存储所有的连接
var clients = [];

var server = net.createServer(function(socket) {

  // socket.setEncoding('utf8');

  // 哪个客户端与我连接socket就是谁
  clients.push(socket);

  console.log("Welcome " + socket.remoteAddress + " to 2080 chatroom");

  // 触发多次
  socket.on('data', clientData).on('error', function(err) {
    clients.splice(clients.indexOf(socket), 1);
    console.log(socket.remoteAddress +"下线了 当前在线" + clients.length);
  });

});

// 广播消息
function boardcast(signal) {
  // console.log(signal);
  // 肯定有用户名和消息
  var username = signal.from;
  var message = signal.message;
  // 我们要发给客户端的东西
  var send = {
    procotol: signal.procotol,
    from: username,
    message: message
  };
  // 广播消息
  clients.forEach(function(client) {
    client.write(JSON.stringify(send));
```

```
  });
}

// 有任何客户端发消息都会触发
function clientData(chunk) {
  // chunk：boardcast|张三|弄啥咧！
  // chunk：{"procotol":"boardcast","from":"张三","message":"弄啥咧！"}
  // chunk：{"procotol":"p2p","from":"张三","to":"李四","message":"弄啥咧！"}
  try {
    var signal = JSON.parse(chunk.toString().trim());
    var procotol = signal.procotol;
    switch (procotol) {
      case 'boardcast':
        boardcast(signal);
        break;
      // case 'p2p':
      //   p2p(signal);
      //   break;
      // case 'shake':
      //   shake(signal);
      //   break;
      default:
        socket.write('弄啥咧！你要干的我干不了');
        break;
    }
  } catch (error) {
    socket.write('弄啥咧！');
  }
}

var port = 2080;
server.listen(port, function(err) {
  if (err) {
    console.log('端口被占用');
    return false;
  }
  console.log("服务端正常启动监听【" + port + "】端口");
});
```

client.js

```

// 客户端

const net = require('net');

const readline = require('readline');
const rl = readline.createInterface(process.stdin, process.stdout);
```

```
rl.question('What is your name? ', function(name) {
  name = name.trim();
  if (!name) {
    throw new Error('没名字还出来混。。');
  }
  // 创建于服务端的连接
  var server = net.connect({ port: 2080, host: '192.168.xx.xx' }, () => {

    console.log("Welcome " + name + " to 2080 chatroom");

    // 监听服务端发过来的数据
    server.on('data', function(chunk) {
      try {
        var signal = JSON.parse(chunk.toString().trim());
        var procotol = signal.procotol;
        switch (procotol) {
          case 'boardcast':
            console.log('\nboardcast[' + signal.from + ']> ' + signal.message +
'\n');
            rl.prompt();
            break;
          default:
            server.write('弄啥咧！你要干的我干不了');
            break;
        }
      } catch (error) {
        server.write('弄啥咧！');
      }
    });

    rl.setPrompt(name + '> '); // 此时没有写到控制台

    rl.prompt(); // 写入控制台

    // 输入一行内容敲回车
    rl.on('line', function(line) {

      // {"procotol":"boardcast","from":"张三","message":"弄啥咧！"}
      var send = {
        procotol: 'boardcast',
        from: name,
        message: line.toString().trim()
      };

      server.write(JSON.stringify(send));

      rl.prompt();

    });
```

```
    rl.on('close', function() {
      // console.log('Have a great day!');
      // process.exit(0);
    });

  });
});
```

coffeescript

##coffeescript

> http://coffeescript.org/

> http://coffee-script.org/

CoffeeScript 尝试用简洁的方式展示 JavaScript 优秀的部分.

##安装

```
sudo npm install -g coffee-script
```

##语法：

```
# 变量赋值
num  = 1
isRight = true
obj =
  a: 5
  b: 'str'
  fun: ->
    @a
console.log(obj.fun())

# 函数定义
func = ->
  obj.a + num

# 函数默认值
func2 = (name, age = 20) ->
  console.log(name + ': ' + age)

# 简单if
str = 'str'
str += 'abc' if isRight

# 标准if..else
if str is 'str'
  str += 'abc'
else if str is 'abc'
  str += 'str'

age = 12
# 范围判断
```

```
isStudent = 10 <age < 25
console.log(isStudent)

# 数组定义
arr = [1, 3, 5, 7, 9]

# 数组遍历
for item in arr
  console.log(item)

# while 循环
console.log(arr.pop()) while arr.length > 0

# 推导
obj2 = {
  key: 'abc'
  value: 'def'
  class: 'This is class'
}
console.log key,value for key, value of obj2 when key isnt 'key'

func2('XiaoMing')

# 自执行函数
do ->
  console.log('Self call.')

nullStr = undefined
if nullStr?
  console.log('nullStr is null')

# 如果null或者undefined, 那么赋值
str ?= 'AA'

func3 = `function abc(){console.log('func3');}`
func3()

try
  `a5 = abc`
catch err
  console.log(err)
finally
  console.log('Finally.')

###
  Multi
###
```

##webStorm中

1.新建 coffeescript File

2.Add Watcher（设置编译）

3.设置Program目录

Gulp-自动化构建工具

##gulp

> http://www.gulpjs.com/

> http://www.gulpjs.com.cn/

> http://www.gulpjs.com.cn/docs/api/

> 前端构建之gulp与常用插件

自动化构建工具

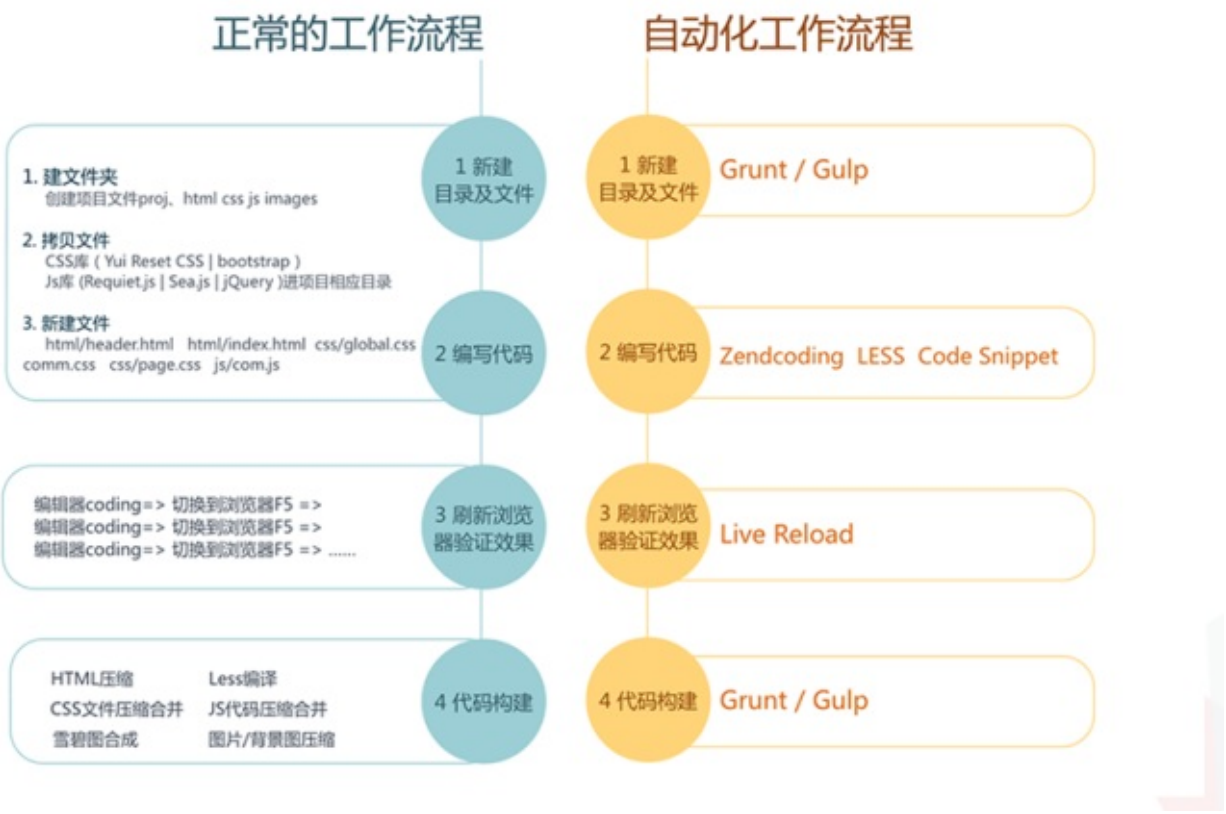
##安装

```
$ npm install gulp -g
```

项目目录再次安装

```
$ npm install gulp
```

##Gulp的作用

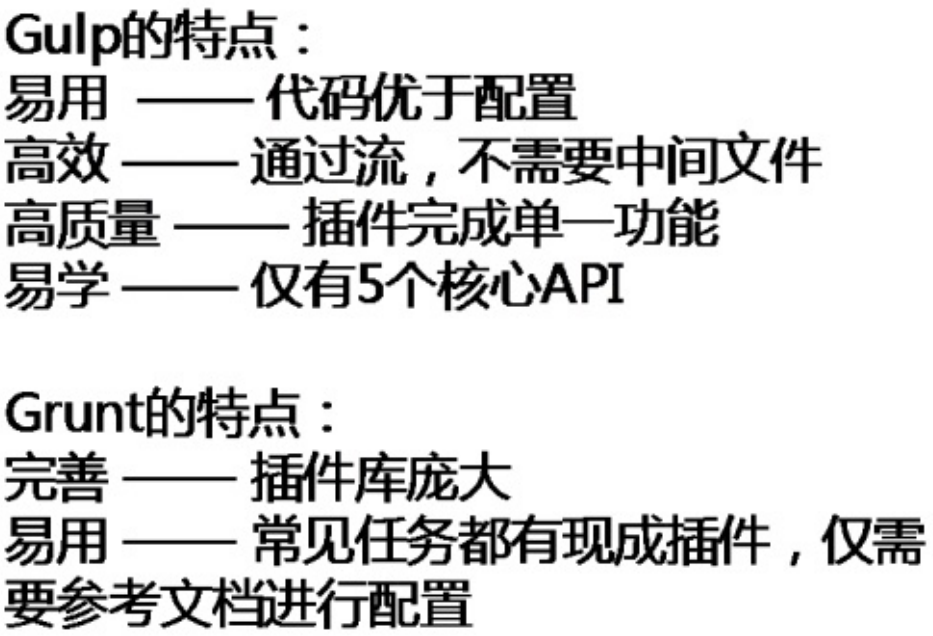


环节	正常工作流	自动化工作流程
1. 新建目录及文件	3-5分钟	3秒
2. 编写代码	全手写	半自动
3. 刷新浏览器验证效果	视项目情况	自动刷新
4. 代码构建	3-5分钟	3秒

##优势

- 入手快，几句代码即可使用
- 不会频繁操作IO，执行速度快
- 插件职责单一，代码质量高
- 仅有5个api

##Gulp与Grunt



##Gulp插件

- 文件合并 gulp-concat
- 文本替换 gulp-replace
- JS压缩 gulp-uglify
- CSS压缩 gulp-cssmin
- 等等

##实例

方括号中的为依赖，会先执行依赖的任务。

```
var gulp = require('gulp');

gulp.task('default', ['copy', 'watch'], function(){
  console.log('default');
});

gulp.task('copy', function(){
  gulp.src('./index.html')
    .pipe(gulp.dest('./dist/'));
});

gulp.task('watch', function(){
  gulp.watch('./index.html', ['copy']);
});
```

顺序:

1.copy task

2.watch task

3.default

然后进入监听状态，循环 1和2

启动

```
//默认回去找default task
gulp

gulp copy

gulp watch
```

##API

- .task 定义任务api
- .src 将文件转换成流
- .dest 将流转换成文件输出
- .watch监控文件变换，则执行XXX

Gulp实现前端构建

##Gulp实现前端构建

1、清空目录

- gulp-clean
- del

2、文件复制

- gulp-copy
- 原生效果达到（gulp.src,gulp.dest）

3、JS压缩

- gulp-uglify

4、CSS压缩

- gulp-minify-css
- gulp-uncss(去除多余的css)

5、文件合并

- gulp-concat

6、启动浏览器

- gulp-open
- browser-sync

7、监视

- gulp-watch
- 原生效果达到（gulp.watch）

8、其他

- yargs 区分不同环境，参数获取方式
- gulp-notify 桌面通知
- gulp-util 日志等通用方法
- run-sequence 控制程序异步还是同步(参数组间都是同步，数组中的为异步)

##安装

```
$ npm install xxx --save-dev
```

##代码

gulpfile.js

```
(function() {
  var browserSync, del, gulp, minifyCss, runSequence, uglify;

  gulp = require('gulp');

  runSequence = require('run-sequence');

  //删除目录
  del = require('del');
  //JS压缩
  uglify = require('gulp-uglify');
  //css压缩
  minifyCss = require('gulp-minify-css');
  //启动浏览器
  browserSync = require('browser-sync').create();

  gulp.task('default', function(callback) {
    return runSequence(['clean'], ['build'], ['serve', 'watch'], callback);
  });
  //删除目录任务
  gulp.task('clean', function(callback) {
    return del(['./dist/'], callback);
  });
  //构建任务，同一数组中异步执行
  gulp.task('build', function(callback) {
    return runSequence(['copy', 'miniJs', 'miniCss'], callback);
  });
  //拷贝任务
  gulp.task('copy', function() {
    return gulp.src('./src/**/*.*').pipe(gulp.dest('./dist/'));
  });
  //压缩js任务
  gulp.task('miniJs', function() {
    return gulp.src('./src/**/*.js').pipe(uglify()).pipe(gulp.dest('./dist/'));
  });
  //压缩css任务
  gulp.task('miniCss', function() {
    return gulp.src('./src/**/*.css').pipe(minifyCss()).pipe(gulp.dest('./dist/'
));
  });
  //文件合并任务
  gulp.task('concat', function() {
    return gulp.src('./src/*.js').pipe(concat('all.js', {
      newLine: ';\n'
    })).pipe(gulp.dest('./dist/'));
  });
  //启动浏览器任务
  gulp.task('serve', function() {
    return browserSync.init({
```

```
      server: {
        baseDir: './dist/'
      },
      port: 7411
    });
  });
  //监控任务，当目录下文件有变化，执行reload任务
  gulp.task('watch', function() {
    return gulp.watch('./src/**/*.*', ['reload']);
  });
  //加载任务，重启浏览器任务
  gulp.task('reload', function(callback) {
    return runSequence(['build'], ['reload-browser'], callback);
  });
  //浏览器加重启任务
  gulp.task('reload-browser', function() {
    return browserSync.reload();
  });

}).call(this);
```

Gulp后端构建

##Gulp后端构建

1、清空目录

- del

2、文件复制

- 原生效果达到（gulp.src,gulp.dest）

3、启动node服务

- gulp-nodemon
- gulp-develop-server

4、监视

- 原生效果达到（gulp.watch）

5、其他

- yargs 区分不同环境，参数获取方式
- gulp-notify 桌面通知
- gulp-util 日志等通用方法
- run-sequence 控制程序异步还是同步(参数组间都是同步，数组中的为异步)

##代码

gulpfile.js

```
(function() {
  var del, developServer, gulp, notify, runSequence;

  gulp = require('gulp');

  del = require('del');

  runSequence = require('run-sequence');

  developServer = require('gulp-develop-server');

  notify = require('gulp-notify');

  gulp.task('default', function(callback) {
    return runSequence(['clean'], ['copyFiles'], ['serve', 'watch'], callback);
  });

  gulp.task('clean', function(callback) {
```

```
    return del('./dist/', callback);
  });

  gulp.task('copyFiles', function() {
    return gulp.src('./src/**/*.js').pipe(gulp.dest('./dist/'));
  });

  gulp.task('serve', function() {
    return developServer.listen({
      path: './dist/index.js'
    });
  });

  gulp.task('watch', function() {
    return gulp.watch('./src/**/*.js', ['reload']);
  });

  gulp.task('reload', function(callback) {
    return runSequence(['copyFiles'], ['reload-node'], callback);
  });

  gulp.task('reload-node', function() {
    developServer.restart();
    return gulp.src('./dist/index.js').pipe(notify('Server restarted ...'));
  });

}).call(this);
```

index.js

```
(function() {
  var app, config, http, onError, onListening, port, server;

  app = require('./libs/app');

  config = require('./config/config');

  http = require('http');

  port = config.port;

  server = http.createServer(app);

  onError = function(error) {
    var bind, ref;
    if (error.syscall !== 'listen') {
      throw error;
    }
    bind = (ref = typeof port === 'string') != null ? ref : 'Pipe ' + {
```

```
      port: 'Port ' + port
    };
    switch (error.code) {
      case 'EACCES':
        console.error(bind + ' requires elevated privileges');
        return process.exit(1);
      case 'EADDRINUSE':
        console.error(bind + ' is already in use');
        return process.exit(1);
      default:
        throw error;
    }
  };

  onListening = function() {
    var addr, bind, ref;
    addr = address();
    bind = (ref = typeof addr === 'string') != null ? ref : 'pipe ' + {
      addr: 'port ' + addr.port
    };
    return debug('Listening on ' + bind);
  };

  server.on('error', onError);

  server.listen(port, function() {
    return console.log('Started...');
  });

}).call(this);
```

config/config.js

```
(function() {
  module.exports = {
    port: 7410
  };

}).call(this);
```

插件

gulp-load-plugins 模块化管理插件

gulp-minify-css 压缩css插件

gulp-sass 将sass预处理为css

gulp-less 将less预处理为css

gulp-sourcemaps 插件

gulp-concat 合并插件

gulp-uglify 压缩JS插件

gulp-util gulp常用工具库插件

yargs插件

gulp-nodemon 自动启动/重启插件

coffee-script 插件

gulp-coffee插件

gulp-livereload 网页自动刷新

gulp-load-plugins 模块化管理插件

##资料

> http://www.qianduancun.com/nodejs/33.html

gulp-minify-css 压缩css插件

##gulp-minify-css

Links：https://www.npmjs.com/package/gulp-minify-css

作用：压缩css。

gulp-sass 将sass预处理为css

##gulp-sass

Links：https://www.npmjs.com/package/gulp-sass

作用：将sass预处理为css

gulp-less 将less预处理为css

##gulp-less

Links：https://www.npmjs.com/package/gulp-less

作用：将less预处理为css

gulp-sourcemaps 插件

##gulp-sourcemaps

Links: https://www.npmjs.com/package/gulp-sourcemaps

作用：处理JS时，生成SourceMap

gulp-concat 合并插件

##gulp-concat

Links: https://www.npmjs.com/package/gulp-concat

作用：合并css，合并js等

gulp-uglify 压缩JS插件

##gulp-uglify

Links: https://www.npmjs.com/package/gulp-uglify

作用：通过UglifyJS来压缩JS文件

gulp-util gulp常用工具库插件

##gulp-util

Links: https://www.npmjs.com/package/gulp-util

作用：gulp常用的工具库

yargs插件

##yargs

Links: https://www.npmjs.com/package/yargs

作用：用于获取启动参数，针对不同参数，切换任务执行过程时需要

gulp-nodemon 自动启动/重启插件

##gulp-nodemon

Links: https://www.npmjs.com/package/gulp-nodemon

作用：自动启动/重启node程序

##资料

gulp-nodemon 和 gulp-livereload 配置

coffee-script 插件

##coffee-script

Links: https://www.npmjs.com/package/coffee-script

作用：gulpfile默认采用js后缀，如果要使用gulpfile.coffee来编写，那么需要此模块

gulp-coffee插件

##gulp-coffee

Links: https://github.com/wearefractal/gulp-coffee

作用：编译coffee代码为Js代码，使用coffeescript必备

gulp-livereload 网页自动刷新

##gulp-livereload 网页自动刷新

资料

Gulp构建前端自动化工作流之：入门介绍及LiveReload的使用

Gulp.js-livereload 不用F5了，实时自动刷新页面来开发

gulp-livereload教程

Moment.js-处理时间插件

##Moment.js

处理时间插件

> http://momentjs.com/
>
> http://momentjs.cn/docs/

##安装

```
npm install moment --save
```

##实例

```
var Moment = require("moment");
console.log(moment().format('D/M/YYYY h:mm:ss'));
```

express 前端框架

##express

> http://www.expressjs.com.cn/starter/generator.html

Async-异步流程控制插件

##Async-异步流程控制

> http://blog.fens.me/nodejs-async/

node-progress进度条插件

##node-progress进度条插件

> https://github.com/visionmedia/node-progress

##实例

```
// node-progress

var ProgressBar = require('progress');

var bar = new ProgressBar('progress: [:bar]', { total: 50, width: 10, complete:
'*' });


var timer = setInterval(function () {
  bar.tick(5);  //进度步长
  if (bar.complete) {
    console.log('\ncomplete\n');
    clearInterval(timer);
  }
}, 100);
```

JSHint-代码规范检查工具

##资料参考

JSHint: 规范团队的JavaScript代码

jshint在gulp中的使用

jshint-stylish 一个外部的报告器

http://www.cnblogs.com/haogj/p/4781677.html

lodash -JavaScript 工具库

##资料

http://lodashjs.com/

资料

资料

JavaScript 标准参考教程（alpha）

官方API

看api注意api稳定性：0，不推荐，2：推荐

Node.js 实战

Node.js API 中文文档

nodejs中文文档

Node.js 教程

##文章

nodejs stream 手册完整中文版本

#package 文件生成

```
npm init
```

npm ls --depth 0

#bower

```
npm install bower -g
bower init
```

框架所用包

##nodejs框架

> https://www.npmjs.com

> package.json详解

##构建npm私有仓库

https://cnpmjs.org

https://npm.taobao.org/

使用CNPM搭建私有NPM仓库

PM2

https://github.com/Unitech/pm2

PM2 使用介绍

pm2 是一个带有负载均衡功能的Node应用的进程管理器.

##cli

###package.json

- alphabetjs

https://www.npmjs.com/package/alphabetjs

alphabetjs是一个小的工具来帮助你输出大的英文字符在控制台/壳或其他平台。

- chalk

https://www.npmjs.com/package/chalk

colors.js曾经是最流行的字符串的造型模块，此处用来输出颜色各样的语句，如红色的错误消息。

- commander

https://www.npmjs.com/package/commander

对Node.js的命令行界面的完整解决方案，灵感来自Ruby的commander

- shelljs

https://www.npmjs.com/package/shelljs

shelljs是一种便携式（Windows / Linux和OS X）的Unix命令Node.js API的实现。你可以用它来消除你的shell脚本依赖Unix而仍然保持其熟悉的和强大的命令。你也可以把它安装在全球范围内，您可以运行它从外部节点项

目说那些粗糙的bash脚本再见！
nodejs 实现封装的shell脚本
##framework
###package.json

- allow-origin

https://www.npmjs.com/package/allow-origin
跨域的快速中间件

- alphabetjs

https://www.npmjs.com/package/alphabetjs
alphabetjs是一个小的工具来帮助你输出大的英文字符在控制台/壳或其他平台。

- body-parser

https://www.npmjs.com/package/body-parser
Node.js体分析中间件。
解析一个中间件请求体在你的处理，有效的req.body物业。
urlencoded

- chalk

https://www.npmjs.com/package/chalk
colors.js曾经是最流行的字符串的造型模块，此处用来输出颜色各样的语句，如红色的错误消息。

- config

https://www.npmjs.com/package/config
配置文件
Node-config组织为您的应用程序部署的层次结构。
它可以让您定义一组默认参数，并将其扩展为不同的部署环境（开发、质量保证、分期、生产等）。

- consolidate

https://www.npmjs.com/package/consolidate
模板引擎整合库

- express

https://www.npmjs.com/package/express

很快，极简的Web框架

- file-stream-rotator

https://www.npmjs.com/package/file-stream-rotator

Nodejs文件流

提供一个基于日期的快速/连接日志的。

- glob

https://www.npmjs.com/package/glob

这是一个全局JavaScript实现。它用MINIMATCH库进行匹配。

- lodash

https://www.npmjs.com/package/lodash

封装的javascript操作

- moment

https://www.npmjs.com/package/moment

一个轻量级的JavaScript日期库解析、验证、操作和格式化日期。

- morgan

https://www.npmjs.com/package/morgan

HTTP请求日志中间件。

- path

https://www.npmjs.com/package/path

这是的Nodejs“路径”模块发布到NPM注册表复制。

- q

https://www.npmjs.com/package/q

promises

- swagger-node-express

https://www.npmjs.com/package/swagger-node-express

API swagger

- swig

https://www.npmjs.com/package/swig

前台模板。

bodyParser

##bodyParser

> https://www.npmjs.com/package/body-parser

##资料

> http://www.tuicool.com/articles/beEJ32a